M&G No. 50037.272US01 / 308287.01

EFFICIENT ALGORITHM AND PROTOCOL FOR REMOTE DIFFERENTIAL COMPRESSION

Field of the Invention

The present invention relates generally to updating data objects over networks with limited bandwidth. More particularly, the present invention relates to a system and method for the differential transfer of object data using a remote differential compression (RDC) methodology. The recursive application of the RDC methodology can be used to further minimize bandwidth usage for the transfer of large objects.

Background of the Invention

The proliferation of networks such as intranets, extranets, and the internet has lead to a large growth in the number of users that share information across wide networks. A maximum data transfer rate is associated with each physical network based on the bandwidth associated with the transmission medium as well as other infrastructure related limitations. As a result of limited network bandwidth, users can experience long delays in retrieving and transferring large amounts of data across the network.

Data compression techniques have become a popular way to transfer large amounts of data across a network with limited bandwidth. Data compression can be generally characterized as either lossless or lossy. Lossless compression involves the transformation of a data set such that an exact reproduction of the data set can be retrieved by applying a decompression transformation. Lossless compression is most often used to compact data, when an exact replica is required.

In the case where the recipient of a data object already has a previous, or older, version of that object, a lossless compression approach called Remote Differential Compression (RDC) may be used to determine and only transfer the differences between the new and the old versions of the object. Since an RDC transfer only involves communicating the observed differences between the new and old

versions (for instance, in the case of files, file modification or last access dates, file attributes, or small changes to the file contents), the total amount of data transferred can be greatly reduced. RDC can be combined with another lossless compression algorithm to further reduce the network traffic. The benefits of RDC are most significant in the case where large objects need to be communicated frequently back and forth between computing devices and it is difficult or infeasible to maintain old copies of these objects, so that local differential algorithms cannot be used.

Summary of the Invention

Briefly stated, the present invention is related to a method and system for updating objects over limited bandwidth networks. Objects are updated between two or more computing devices using remote differential compression (RDC) techniques such that required data transfers are minimized. In one aspect, efficient large object transfers are achieved by recursively applying the RDC algorithm to its own metadata; a single or multiple recursion step(s) may be used in this case to reduce the amount of metadata sent over the network by the RDC algorithm. Objects and/or signature and chunk length lists can be chunked by locating boundaries at dynamically determined locations. A mathematical function evaluates hash values associated within a horizon window relative to potential chunk boundary. The described method and system is useful in a variety of networked applications, such as peer-to-peer replicators, email clients and servers, client-side caching systems, general-purpose copy utilities, database replicators, portals, software update services, file/data synchronization, and others.

A more complete appreciation of the present invention and its improvements can be obtained by reference to the accompanying drawings, which are briefly summarized below, to the following detailed description of illustrative embodiments of the invention, and to the appended claims.

Brief Description of the Drawings

Non-limiting and non-exhaustive embodiments of the present invention are described with reference to the following drawings.

FIG. 1 is a diagram illustrating an operating environment;

FIG. 2 is a diagram illustrating an example computing device;

FIGS. 3A and 3B are diagrams illustrating an example RDC procedure;

FIGS. 4A and 4B are diagrams illustrating process flows for the interaction between a local device and a remote device during an example RDC procedure;

FIGS. 5A and 5B are diagrams illustrating process flows for recursive remote differential compression of the signature and chunk length lists in an example interaction during an RDC procedure;

FIG. 6 is a diagram that graphically illustrates an example of recursive compression in an example RDC sequence;

FIG. 7 is a diagram illustrating the interaction of a client and server application using an example RDC procedure;

FIG. 8 is a diagram illustrating a process flow for an example chunking procedure;

FIG. 9 is a diagram of example instruction code for an example chunking procedure;

FIGS. 10A and 10B are diagrams of another example instruction code for another example chunking procedure, arranged according to at least one aspect of the present invention.

Detailed Description of the Preferred Embodiment

Various embodiments of the present invention will be described in detail with reference to the drawings, where like reference numerals represent like parts and assemblies throughout the several views. Reference to various embodiments does not limit the scope of the invention, which is limited only by the scope of the claims attached hereto. Additionally, any examples set forth in this specification are not intended to be limiting and merely set forth some of the many possible embodiments for the claimed invention.

The present invention is described in the context of local and remote computing devices (or "devices", for short) that have one or more commonly associated objects stored thereon. The terms "local" and "remote" refer to one instance of the method. However, the same device may play both a "local" and a "remote" role in different instances. Remote Differential Compression (RDC) methods are used to efficiently update the commonly associated objects over a network with limited-bandwidth. When a device having a new copy of an object needs to update a device having an older copy of the same object, or of a similar object, the RDC method is employed to only transmit the differences between the objects over the network. An example described RDC method uses (1) a recursive approach for the transmission of the RDC metadata, to reduce the amount of metadata transferred for large objects, and (2) a local maximum-based chunking method to increase the precision associated with the object differencing such that bandwidth utilization is minimized. Some example applications that benefit from the described RDC methods include: peer-to-peer replication services, file-transfer protocols such as SMB, virtual servers that transfer large images, email servers, cellular phone and PDA synchronization, database server replication, to name just a few.

Operating Environment

FIG. 1 is a diagram illustrating an example operating environment for the present invention. As illustrated in the figure, devices are arranged to communicate over a network. These devices may be general purpose computing device, special purpose computing devices, or any other appropriate devices that are connected to a network. The network 102 may correspond to any connectivity topology including, but not limited to: a direct wired connection (e.g., parallel port, serial port, USB, IEEE 1394, etc), a wireless connection (e.g., IR port, Bluetooth port, etc.), a wired network, a wireless network, a local area network, a wide area network, an ultra-wide area network, an internet, an intranet, and an extranet.

In an example interaction between device A (100) and device B (101), different versions of an object are locally stored on the two devices: object O_A on 100

and object O_B on 101. At some point, device A (100) decides to update its copy of object O_A with the copy (object O_B) stored on device B (101), and sends a request to device B (101) to initiate the RDC method. In an alternate embodiment, the RDC method could be initiated by device B (101).

Device A (100) and device B (101) both process their locally stored object and divide the associated data into a variable number of chunks in a data-dependent fashion (e.g., chunks 1 – n for object O_B, and chunks 1 – k for object O_A, respectively). A set of signatures such as strong hashes (SHA) for the chunks are computed locally by both the devices. The devices both compile separate lists of the signatures. During the next step of the RDC method, device B (101) transmits its computed list of signatures and chunk lengths 1 – n to device A (100) over the network 102. Device A (100) evaluates this list of signatures by comparing each received signature to its own generated signature list 1 – k. Mismatches in the signature lists indicate one or more differences in the objects that require correction. Device A (100) transmits a request for device B (101) to send the chunks that have been identified by the mismatches in the signature lists. Device B (101) subsequently compresses and transmits the requested chunks, which are then reassembled by device A (100) after reception and decompression are accomplished. Device A (100) reassembles the received chunks together with its own matching chunks to obtain a local copy of object O_B.

Example Computing Device

FIG. 2 is a block diagram of an example computing device that is arranged in accordance with the present invention. In a basic configuration, computing device 200 typically includes at least one processing unit (202) and system memory (204). Depending on the exact configuration and type of computing device, system memory 204 may be volatile (such as RAM), non-volatile (such as ROM, flash memory, etc.) or some combination of the two. System memory 204 typically includes an operating system (205); one or more program modules (206); and may include

program data (207). This basic configuration is illustrated in FIG. 2 by those components within dashed line 208.

Computing device 200 may also have additional features or functionality. For example, computing device 200 may also include additional data storage devices (removable and/or non-removable) such as, for example, magnetic disks, optical disks, or tape. Such additional storage is illustrated in FIG. 2 by removable storage 209 and non-removable storage 210. Computer storage media may include volatile and non-volatile, removable and non-removable media implemented in any method or technology for storage of information, such as computer readable instructions, data structures, program modules or other data. System memory 204, removable storage 209 and non-removable storage 210 are all examples of computer storage media. Computer storage media includes, but is not limited to, RAM, ROM, EEPROM, flash memory or other memory technology, CD-ROM, digital versatile disks (DVD) or other optical storage, magnetic cassettes, magnetic tape, magnetic disk storage or other magnetic storage devices, or any other medium which can be used to store the desired information and which can be accessed by computing device 200. Any such computer storage media may be part of device 200. Computing device 200 may also have input device(s) 212 such as keyboard, mouse, pen, voice input device, touch input device, etc. Output device(s) 214 such as a display, speakers, printer, etc. may also be included. All these devices are known in the art and need not be discussed at length here.

Computing device 200 also contains communications connection(s) 216 that allow the device to communicate with other computing devices 218, such as over a network. Communications connection(s) 216 is an example of communication media. Communication media typically embodies computer readable instructions, data structures, program modules or other data in a modulated data signal such as a carrier wave or other transport mechanism and includes any information delivery media. The term "modulated data signal" means a signal that has one or more of its characteristics set or changed in such a manner as to encode information in the signal. By way of example, and not limitation, communication media includes wired media such as a

wired network or direct-wired connection, and wireless media such as acoustic, RF, microwave, satellite, infrared and other wireless media. The term computer readable media as used herein includes both storage media and communication media.

Various procedures and interfaces may be implemented in one or more application programs that reside in system memory 204. In one example, the application program is a remote differential compression algorithm that schedules file synchronization between the computing device (e.g., a client) and another remotely located computing device (e.g., a server). In another example, the application program is a compression/decompression procedure that is provided in system memory 204 for compression and decompressing data. In still another example, the application program is a decryption procedure that is provided in system memory 204 of a client device.

Remote Differential Compression (RDC)

FIGS. 3A and 3B are diagrams illustrating an example RDC procedure according to at least one aspect of the present invention. The number of chunks in particular can vary for each instance depending on the actual objects O_A and O_B.

Referring to FIG. 3A, the basic RDC protocol is negotiated between two computing devices (device A and device B). The RDC protocol assumes implicitly that the devices A and B have two different instances (or versions) of the same object or resource, which are identified by object instances (or versions) O_A and O_B, respectively. For the example illustrated in this figure, device A has an old version of the resource O_A, while device B has a version O_B with a slight (or incremental) difference in the content (or data) associated with the resource.

The protocol for transferring the updated object O_B from device B to device A is described below. A similar protocol may be used to transfer an object from device A to device B, and that the transfer can be initiated at the behest of either device A or device B without significantly changing the protocol described below.

1. Device A sends device B a request to transfer Object O_B using the RDC protocol. In an alternate embodiment, device B initiates the

transfer; in this case, the protocol skips step 1 and starts at step 2 below.

2. Device A partitions Object O_A into chunks 1 – k, and computes a signature Sig_{Ai} and a length (or size in bytes) Len_{Ai} for each chunk 1…k of Object O_A. The partitioning into chunks will be described in detail below. Device A stores the list of signatures and chunk lengths ((Sig_{A1}, Len_{A1}) ... (Sig_{Ak}, Len_{Ak})).

3. Device B partitions Object O_B into chunks 1 – n, and computes a signature Sig_{Bi} and a length Len_{Bi} for each chunk 1…n of Object O_B. The partitioning algorithm used in step 3 must match the one in step 2 above.

4. Device B sends a list of its computed chunk signatures and chunk lengths ((Sig_{B1}, Len_{B1}) ... (Sig_{Bn}, Len_{Bn})) that are associated with Object O_B to device A. The chunk length information may be subsequently used by device A to request a particular set of chunks by identifying them with their start offset and their length. Because of the sequential nature of the list, it is possible to compute the starting offset in bytes of each chunk Bi by adding up the lengths of all preceding chunks in the list.

   In another embodiment, the list of chunk signatures and chunk lengths is compactly encoded and further compressed using a lossless compression algorithm before being sent to device A.

5. Upon receipt of this data, device A compares the received signature list against the signatures Sig_{A1} … Sig_{Ak} that it computed for Object O_A in step 2, which is associated with the old version of the content.

6. Device A sends a request to device B for all the chunks whose signatures received in step 4 from device B failed to match any of the signatures computed by device A in step 2. For each requested

chunk Bi, the request comprises the chunk start offset computed by device A in step 4 and the chunk length.

7. Device B sends the content associated with all the requested chunks to device A. The content sent by device B may be further compressed using a lossless compression algorithm before being sent to device A.

8. Device A reconstructs a local copy of Object O_B by using the chunks received in step 7 from device B, as well as its own chunks of Object O_A that matched signatures sent by device B in step 4. The order in which the local and remote chunks are rearranged on device A is determined by the list of chunk signatures received by device A in step 4.

The partitioning steps 2 and 3 may occur in a data-dependent fashion that uses a fingerprinting function that is computed at every byte position in the associated object (O_A and O_B, respectively). For a given position, the fingerprinting function is computed using a small data window surrounding that position in the object; the value of the fingerprinting function depends on all the bytes of the object included in that window. The fingerprinting function can be any appropriate function, such as, for example, a hash function or a Rabin polynomial.

Chunk boundaries are determined at positions in the Object for which the fingerprinting function computes to a value that satisfies a chosen condition. The chunk signatures may be computed using a cryptographically secure hash function (SHA), or some other hash function such as a collision-resistant hash function.

The signature and chunk length list sent in step 4 provides a basis for reconstructing the object using both the original chunks and the identified updated or new chunks. The chunks that are requested in step 6 are identified by their offset and

lengths. The object is reconstructed on device A by using local and remote chunks whose signatures match the ones received by device A in step 4, in the same order.

After the reconstruction step is completed by device A, Object O_A can be deleted and replaced by the copy of Object O_B that was reconstructed on device A. In other embodiments, device A may keep Object O_A around for potential "reuse" of chunks during future RDC transfers.

For large objects, the basic RDC protocol instance illustrated in FIG. 3A incurs a significant fixed overhead in Step 4, even if Object O_A and Object O_B are very close, or identical. Given an average chunk size C, the amount of information transmitted over the network in Step 4 is proportional to the size of Object O_B, specifically it is proportional to the size of Object O_B divided by C, which is the number of chunks of Object B, and thus of (chunk signature, chunk length) pairs transmitted in step 4.

For example, referring to FIG. 6, a large image (e.g., a virtual hard disk image used by a virtual machine monitor such as, for example, Microsoft Virtual Server) may result in an Object (O_B) with a size of 9.1GB. For an average chunk size C equal to 3KB, the 9GB object may result in 3 million chunks being generated for Object O_B, with 42MB of associated signature and chunk length information that needs to be sent over the network in Step 4. Since the 42MB of signature information must be sent over the network even when the differences between Object O_A and Object O_B (and thus the amount of data that needs to be sent in Step 7) are very small, the fixed overhead cost of the protocol is excessively high.

This fixed overhead cost can be significantly reduced by using a recursive application of the RDC protocol instead of the signature information transfer in step 4. Referring to FIG. 3B, additional steps 4.2 – 4.8 are described as follows below that replace step 4 of the basic RDC algorithm. Steps 4.2 – 4.8 correspond to a recursive application of steps 2 – 8 of the basic RDC protocol described above. The recursive application can be further applied to step 4.4 below, and so on, up to any desired recursion depth.

4.2. Device A performs a recursive chunking of its signature and chunk length list ($(Sig_{A1}, Len_{A1}) \ldots (Sig_{Ak}, Len_{Ak})$) into recursive signature chunks, obtaining another list of recursive signatures and recursive chunk lengths ($(RSig_{A1}, RLen_{A1}) \ldots (RSig_{As}, RLen_{As})$), where $s << k$.

4.3. Device B recursively chunks up the list of signatures and chunk lengths ($(Sig_{B1}, Len_{B1}) \ldots (Sig_{Bn}, Len_{Bn})$) to produce a list of recursive signatures and recursive chunk lengths ($(RSig_{B1}, RLen_{B1}) \ldots (RSig_{Br}, RLen_{Br})$), where $r << n$.

4.4. Device B sends an ordered list of recursive signatures and recursive chunk lengths ($(RSig_{B1}, RLen_{B1}) \ldots (RSig_{Br}, RLen_{Br})$) to device A. The list of recursive chunk signatures and recursive chunk lengths is compactly encoded and may be further compressed using a lossless compression algorithm before being sent to device A.

4.5. Device A compares the recursive signatures received from device B with its own list of recursive signatures computed in Step 4.2.

4.6. Device A sends a request to device B for every distinct recursive signature chunk (with recursive signature $RSig_{Bk}$) for which device A does not have a matching recursive signature in its set ($RSig_{A1} \ldots RSig_{As}$).

4.7. Device B sends device A the requested recursive signature chunks. The requested recursive signature chunks may be further compressed using a lossless compression algorithm before being sent to device A.

4.8. Device A reconstructs the list of signatures and chunk information ($(Sig_{B1}, Len_{B1}) \ldots (Sig_{Bn}, Len_{Bn})$) using the locally matching recursive signature chunks, and the recursive chunks received from device B in Step 4.7.

After step 4.8 above is completed, execution continues at step 5 of the basic RDC protocol described above, which is illustrated in FIG. 3A.

As a result of the recursive chunking operations, the number of recursive signatures associated with the objects is reduced by a factor equal to the average chunk size C, yielding a significantly smaller number of recursive signatures ($r << n$ for object O_A and $s << k$ for object O_B, respectively). In one embodiment, the same chunking parameters could be used for chunking the signatures as for chunking the original objects O_A and O_B. In an alternate embodiment, other chunking parameters may be used for the recursive steps.

For very large objects the above recursive steps can be applied k times, where $k \geq 1$. For an average chunk size of C, recursive chunking may reduce the size of the signature traffic over the network (steps 4.2 through 4.8) by a factor approximately corresponding to C^k. Since C is relatively large, a recursion depth of greater than one may only be necessary for very large objects.

In one embodiment, the number of recursive steps may be dynamically determined by considering parameters that include one or more of the following: the expected average chunk size, the size of the objects O_A and/or O_B, the data format of the objects O_A and/or O_B, the latency and bandwidth characteristics of the network connecting device A and device B.

The fingerprinting function used in step 2 is matched to the fingerprinting function that is used in step 3. Similarly, the fingerprinting function used in step 4.2 is matched to the fingerprinting function that is used in step 4.3. The fingerprinting function from steps 2 - 3 can optionally be matched to the fingerprinting function from steps 4.2 - 4.3.

As described previously, each fingerprinting function uses a small data window that surrounds a position in the object; where the value associated with the fingerprinting function depends on all the bytes of the object that are included inside the data window. The size of the data window can be dynamically adjusted based on one or more criteria. Furthermore, the chunking procedure uses the value of the fingerprinting function and one or more additional chunking parameters to determine the chunk boundaries in steps 2 – 3 and 4.2 – 4.3 above.

By dynamically changing the window size and the chunking parameters, the chunk boundaries are adjusted such that any necessary data transfers are accomplished with minimal consumption of the available bandwidth.

Example criteria for adjusting the window size and the chunking parameters include: a data type associated with the object, environmental constraints, a usage model, the latency and bandwidth characteristics of the network connecting device A and device B, and any other appropriate model for determining average data transfer block sizes. Example data types include word processing files, database images, spreadsheets, presentation slide shows, and graphic images. An example usage model may be where the average number of bytes required in a typical data transfer is monitored.

Changes to a single element within an application program can result in a number of changes to the associated datum and/or file. Since most application programs have an associated file type, the file type is one possible criteria that is worthy of consideration in adjusting the window size and the chunking parameters. In one example, the modification of a single character in a word processing document results in approximately 100 bytes being changed in the associated file. In another example, the modification of a single element in a database application results in 1000 bytes being changed in the database index file. For each example, the appropriate window size and chunking parameters may be different such that the chunking procedure has an appropriate granularity that is optimized based on the particular application.

Example Process Flow

FIGS. 4A and 4B are diagrams illustrating process flows for the interaction between a local device (e.g., device A) and a remote device (e.g., device B) during an example RDC procedure that is arranged in accordance with at least one aspect of the present invention. The left hand side of FIG. 4A illustrates steps 400 – 413 that are operated on the local device A, while the right hand side of FIG. 4A illustrates steps 450 – 456 that are operated on the remote device B.

As illustrated in FIG. 4A, the interaction starts by device A requesting an RDC transfer of object O_B in step 400, and device B receiving this request in step 450. Following this, both the local device A and remote device B independently compute fingerprints in steps 401 and 451, divide their respective objects into chunks in steps 402 and 452, and compute signatures (e.g., SHA) for each chunk in steps 403 and 453, respectively.

In step 454, device B sends the signature and chunk length list computed in steps 452 and 453 to device A, which receives this information in step 404.

In step 405, the local device A initializes the list of requested chunks to the empty list, and initializes the tracking offset for the remote chunks to 0. In step 406, the next (signature, chunk length) pair (Sig_{Bi}, Len_{Bi}) is selected for consideration from the list received in step 404. In step 407, device A checks whether the signature Sig_{Bi} selected in step 406 matches any of the signatures it computed during step 403. If it matches, execution continues at step 409. If it doesn't match, the tracking remote chunk offset and the length in bytes Len_{Bi} are added to the request list in step 408. At step 409, the tracking offset is incremented by the length of the current chunk Len_{Bi}.

In step 410, the local device A tests whether all (signature, chunk length) pairs received in step 404 have been processed. If not, execution continues at step 406. Otherwise, the chunk request list is suitably encoded in a compact fashion, compressed, and sent to the remote device B at step 411.

The remote device B receives the compressed list of chunks at step 455, decompresses it, then compresses and sends back the chunk data at step 456.

The local device receives and decompresses the requested chunk data at step 412. Using the local copy of the object O_A and the received chunk data, the local devices reassembles a local copy of O_B at step 413.

FIG. 4B illustrates a detailed example for step 413 from FIG. 4A. Processing continues at step 414, where the local device A initializes the reconstructed object to empty.

In step 415, the next (signature, chunk length) pair (Sig_{Bi}, Len_{Bi}) is selected for consideration from the list received in step 404. In step 416, device A

checks whether the signature Sig_{Bi} selected in step 417 matches any of the signatures it computed during step 403.

If it matches, execution continues at step 417, where the corresponding local chunk is appended to the reconstructed object. If it doesn't match, the received and decompressed remote chunk is appended to the reconstructed object in step 418.

In step 419, the local device A tests whether all (signature, chunk length) pairs received in step 404 have been processed. If not, execution continues at step 415. Otherwise, the reconstructed object is used to replace the old copy of the object O_A on device A in step 420.

Example Recursive Signature Transfer Process Flow

FIGS. 5A and 5B are diagrams illustrating process flows for recursive transfer of the signature and chunk length list in an example RDC procedure that is arranged according to at least one aspect of the present invention. The below described procedure may be applied to both the local and remote devices that are attempting to update commonly associated objects.

The left hand side of FIG. 5A illustrates steps 501 – 513 that are operated on the local device A, while the right hand side of FIG. 5A illustrates steps 551 – 556 that are operated on the remote device B. Steps 501 – 513 replace step 404 in FIG. 4A while steps 551 – 556 replace step 454 in FIG. 4A.

In steps 501 and 551, both the local device A and remote device B independently compute recursive fingerprints of their signature and chunk length lists $((Sig_{A1},Len_{A1}), ... (Sig_{Ak},Len_{Ak}))$ and $((Sig_{B1},Len_{B1}), ... (Sig_{Bn},Len_{Bn}))$, respectively, that had been computed in steps 402/403 and 452/453, respectively. In steps 502 and 552 the devices divide their respective signature and chunk length lists into recursive chunks, and in steps 503 and 553 compute recursive signatures (e.g., SHA) for each recursive chunk, respectively.

In step 554, device B sends the recursive signature and chunk length list computed in steps 552 and 553 to device A, which receives this information in step 504.

In step 505, the local device A initializes the list of requested recursive chunks to the empty list, and initializes the tracking remote recursive offset for the remote recursive chunks to 0. In step 506, the next (recursive signature, recursive chunk length) pair ($RSig_{Bi}$, $RLen_{Bi}$) is selected for consideration from the list received in step 504. In step 507, device A checks whether the recursive signature $RSig_{Bi}$ selected in step 506 matches any of the recursive signatures it computed during step 503. If it matches, execution continues at step 509. If it doesn't match, the tracking remote recursive chunk offset and the length in bytes $RLen_{Bi}$ are added to the request list in step 508. At step 509, the tracking remote recursive offset is incremented by the length of the current recursive chunk $RLen_{Bi}$.

In step 510, the local device A tests whether all (recursive signature, recursive chunk length) pairs received in step 504 have been processed. If not, execution continues at step 506. Otherwise, the recursive chunk request list is compactly encoded, compressed, and sent to the remote device B at step 511.

The remote device B receives the compressed list of recursive chunks at step 555, uncompressed the list, then compresses and sends back the recursive chunk data at step 556.

The local device receives and decompresses the requested recursive chunk data at step 512. Using the local copy of the signature and chunk length list ((Sig_{A1},Len_{A1}), ... (Sig_{Ak},Len_{Ak})) and the received recursive chunk data, the local devices reassembles a local copy of the signature and chunk length list ((Sig_{B1},Len_{B1}), ... (Sig_{Bk},Len_{Bn})) at step 513. Execution then continues at step 405 in FIG. 4A.

FIG. 5B illustrates a detailed example for step 513 from FIG. 5A. Processing continues at step 514, where the local device A initializes the list of remote signatures and chunk lengths, SIGCL, to the empty list.

In step 515, the next (recursive signature, recursive chunk length) pair ($RSig_{Bi}$, $RLen_{Bi}$) is selected for consideration from the list received in step 504. In step 516, device A checks whether the recursive signature $RSig_{Bi}$ selected in step 515 matches any of the recursive signatures it computed during step 503.

If it matches, execution continues at step 517, where device A appends the corresponding local recursive chunk to SIGCL. If it doesn't match, the remote received recursive chunk is appended to SIGCL at step 518.

In step 519, the local device A tests whether all (recursive signature, recursive chunk length) pairs received in step 504 have been processed. If not, execution continues at step 515. Otherwise, the local copy of the signature and chunk length list ((Sig_{B1},Len_{B1}), ... (Sig_{Bk},Len_{Bn})) is set to the value of SIGCL in step 520. Execution then continues back to step 405 in FIG. 4A.

The recursive signature and chunk length list may optionally be evaluated to determine if additional recursive remote differential compression is necessary to minimize bandwidth utilization as previously described. The recursive signature and chunk length list can be recursively compressed using the described chunking procedure by replacing steps 504 and 554 with another instance of the RDC procedure, and so on, until the desired compression level is achieved. After the recursive signature list is sufficiently compressed, the recursive signature list is returned for transmission between the remote and local devices as previously described.

FIG. 6 is a diagram that graphically illustrates an example of recursive compression in an example RDC sequence that is arranged in accordance with an example embodiment. For the example illustrated in FIG. 6, the original object is 9.1GB of data. A signature and chunk length list is compiled using a chunking procedure, where the signature and chunk length list results in 3 million chunks (or a size of 42MB). After a first recursive step, the signature list is divided into 33 thousand chunks and reduced to a recursive signature and recursive chunk length list with size 33KB. By recursively compressing the signature list, bandwidth utilization for transferring the signature list is thus dramatically reduced, from 42MB to about 395KB.

Example Object Updating

FIG. 7 is a diagram illustrating the interaction of a client and server application using an example RDC procedure that is arranged according to at least one

aspect of the present invention. The original file on both the server and the client contained text "The quick fox jumped over the lazy brown dog. The dog was so lazy that he didn't notice the fox jumping over him."

At a subsequent time, the file on the server is updated to: "The quick fox jumped over the lazy brown dog. The **brown** dog was so lazy that he didn't notice the fox jumping over him."

As described previously, the client periodically requests the file to be updated. The client and server both chunk the object (the text) into chunks as illustrated. On the client, the chunks are: "The quick fox jumped", "over the lazy brown dog.", "The dog was so lazy that he didn't notice", and "the fox jumping over him."; the client signature list is generated as: SHA_{11}, SHA_{12}, SHA_{11}, and SHA_{14}. On the server, the chunks are: "The quick fox jumped", "over the lazy brown dog.", "The brown dog was", "so lazy that he didn't notice", and "the fox jumping over him." ; the server signature list is generated as: SHA_{21}, SHA_{22}, SHA_{23}, SHA_{24}, and SHA_{25}.

The server transmits the signature list (SHA_{21} - SHA_{25}) using a recursive signature compression technique as previously described. The client recognizes that the locally stored signature list (SHA_{11}- SHA_{14}) does not match the received signature list (SHA_{21} - SHA_{25}), and requests the missing chunks 3 and 4 from the server. The server compresses and transmits chunks 3 and 4 ("The brown dog was", and "so lazy that he didn't notice"). The client receives the compressed chunks, decompresses them, and updates the file as illustrated in FIG. 7.

Chunking Analysis

The effectiveness of the basic RDC procedure described above may be increased by optimizing the chunking procedures that are used to chunk the object data and/or chunk the signature and chunk length lists.

The basic RDC procedure has a network communication overhead cost that is identified by the sum of:

(S1) |Signatures and chunk lengths from B| = $|O_B|$ * |SigLen| / C,

where $|O_B|$ is the size in bytes of Object O_B, SigLen is the size in bytes of a (signature, chunk length) pair, and C is the expected average chunk size in bytes; and

(S2) $\sum$ *chunk_length*, where (*signature*, *chunk_length*) $\in$ Signatures from B,

and *signature* $\notin$ Signatures from A

The communication cost thus benefits from a large average chunk size and a large intersection between the remote and local chunks. The choice of how objects are cut into chunks determines the quality of the protocol. The local and remote device must agree, without prior communication, on where to cut an object. The following describes and analyzes various methods for finding cuts.

The following characteristics are assumed to be known for the cutting algorithm:

1. Slack: The number of bytes required for chunks to reconcile between file differences. Consider sequences s1, s2, and s3, and form the two sequences s1s3, s2s3 by concatenation. Generate the chunks for those two sequences Chunks1, and Chunks2. If Chunks1' and Chunks2' are the sums of the chunk lengths from Chunks1 and Chunks2, respectively, until the first common suffix is reached, the slack in bytes is given by the following formula:

$$\text{slack} = \text{Chunks}_1' - |s_1| = \text{Chunks}_2' - |s_2|$$

2. Average chunk size C:

When Objects O_A and O_B have S segments in common with average size K, the number of chunks that can be obtained locally on the client is given by:

$$S * \lfloor (K - \text{slack})/C \rfloor$$

and (S2) above rewrites to:

$$|O_A| - S * \lfloor (K - slack)/C \rfloor$$

Thus, a chunking algorithm that minimizes slack will minimize the number of bytes sent over the wire. It is therefore advantageous to use chunking algorithms that minimize the expected slack.

Fingerprinting Functions

All chunking algorithms use a fingerprinting function, or hash, that depends on a small window, that is, a limited sequence of bytes. The execution time of the hash algorithms used for chunking is independent of the hash window size when those algorithms are amenable to finite differencing (strength reduction) optimizations. Thus, for a hash window of size k it is should be easy (require only a constant number of steps) to compute the hash $\#[b_1,\ldots,b_{k-1},b_k]$ using b_0 , b_k, and $\#[b_0,b_1,\ldots,b_{k-1}]$ only. Various hashing functions can be employed such as hash functions using Rabin polynomials, as well as other hash functions that appear computationally more efficient based on tables of pre-computed random numbers.

In one example, a 32 bit Adler hash based on the rolling checksum can be used as the hashing function for fingerprinting. This procedure provides a reasonably good random hash function by using a fixed table with 256 entries, each a precomputed 16 bit random number. The table is used to convert fingerprinted bytes into a random 16 bit number. The 32 bit hash is split into two 16 bit numbers sum1 and sum2, which are updated given the procedure:

$$sum1 \mathrel{+}= table[b_k] - table[b_0]$$

$$sum2 \mathrel{+}= sum1 - k^* table[b_0]$$

In another example, a 64 bit random hash with cyclic shifting may be used as the hashing function for fingerprinting. The period of a cyclic shift is bounded by the size of the hash value. Thus, using a 64 bit hash value sets the period of the hash to 64. The procedure for updating the hash is given as:

hash = hash ^ ((table[b_0] << l) | (table[b_0] >> u)) ^ table[b_k];

hash = (hash << 1) | (hash >> 63);

where l = k % 64 and u = 64 - l

In still another example, other shifting methods may be employed to provide fingerprinting. Straight forward cyclic shifting produces a period of limited length, and is bounded by the size of the hash value. Other permutations have longer periods. For instance, the permutation given by the cycles (1 2 3 0) (5 6 7 8 9 10 11 12 13 14 4) (16 17 18 19 20 21 15) (23 24 25 26 22) (28 29 27) (31 30) has a period of length 4*3*5*7*11 = 4620. The single application of this example permutation can be computed using a right shift followed by operations that patch up the positions at the beginning of each interval.

Analysis of previous art for chunking at pre-determined patterns

Previous chunking methods are determined by computing a fingerprinting hash with a pre-determined window size k (= 48), and identifying cut points based on whether a subset of the hash bits match a pre-determined pattern. With random hash values, this pattern may as well be 0, and the relevant subset may as well be a prefix of the hash. In basic instructions, this translates to a predicate of the form:

CutPoint(hash) $\equiv$ 0 = = (hash & ((1 << c) -1)),

where c is the number of bits that are to be matched against.

Since the probability for a match given a random hash function is 2^{-c}, an average chunk size $C = 2^c$ results. However, neither the minimal, nor the maximal chunk size is determined by this procedure. If a minimal chunk length of *m* is imposed, then the average chunk size is:

$$C = m + 2^c$$

A rough estimate of the expected slack is obtained by considering streams s_1s_3 and s_2s_3. Cut points in s_1 and s_2 may appear at arbitrary places. Since the average chunk length is $C = m + 2^c$, about $(2^c/C)^2$ of the last cut-points in s_1 and s_2 will be beyond distance m. They will contribute to slack at around 2^c. The remaining $1 - (2^c/C)^2$ contribute with slack of length about C. The expected slack will then be around $(2^c/C)^3 + (1 - (2^c/C)^2)*(C/C) = (2^c/C)^3 + 1 - (2^c/C)^2$, which has global minimum for $m = 2^{c-1}$, with a value of about 23/27 = 0.85. A more precise analysis gives a somewhat lower estimate for the remaining $1 - (2^c/C)^2$ fraction, but will also need to compensate for cuts within distance *m* inside s_3, which contributes to a higher estimate.

Thus, the expected slack for the prior art is approximately 0.85 * C.

Chunking at Filters (New Art)

Chunking at filters is based on fixing a filter, which is a sequence of patterns of length *m*, and matching the sequence of fingerprinting hashes against the filter. When the filter does not allow a sequence of hashes to match both a prefix and a suffix of the filter it can be inferred that the minimal distance between any two matches must be at least *m*. An example filter may be obtained from the CutPoint predicate used in the previous art, by setting the first *m* - 1 patterns to

$$0 \mathrel{!}= (\text{hash} \;\&\; ((1 << c) - 1))$$

and the last pattern to:

$$0 == (\text{hash} \;\&\; ((1 << c) - 1)).$$

The probability for matching this filter is given by $(1 - p)^{m-1} p$ where *p* is 2^{-c}. One may compute that the expected chunk length is given by the inverse of the probability for matching a filter (it is required that the filter not allow a sequence to

match both a prefix and suffix), thus the expected length of the example filter is $(1 - p)^{-m+1} p^{-1}$. This length is minimized when setting p := $1/m$, and it turns out to be around (e * m). The average slack hovers around 0.8, as can be verified by those skilled in the art. An alternative embodiment of this method uses a pattern that works directly with the raw input and does not use rolling hashes.

Chunking at Local Maxima (New Art)

Chunking at Local Maxima is based on choosing as cut points positions that are maximal within a bounded horizon. In the following, we shall use h for the value of the horizon. We say that the hash at position *offset* is an h-local maximum if the hash values at offsets *offset-h, ..., offset-1,* as well as *offset+1, ..., offset+h* are all smaller than the hash value at *offset.* In other words, all positions h steps to the left and h steps to the right have lesser hash values. Those skilled in the art will recognize that local maxima may be replaced by local minima or any other metric based comparison (such as "*closest to the median hash value*").

The set of local maxima for an object of size n may be computed in time bounded by 2•n operations such that the cost of computing the set of local maxima is close to or the same as the cost of computing the cut-points based on independent chunking. Chunks generated using local maxima always have a minimal size corresponding to h, with an average size of approximately $2h+1$. A CutPoint procedure is illustrated in FIGS.8 and 9, and is described as follows below:

1. Allocate an array M of length h whose entries are initialized with the record {isMax=false, hash=0, offset=0}. The first entry in each field (isMax) indicates whether a candidate can be a local maximum. The second field entry (hash) indicates the hash value associated with that entry, and is initialized to 0 (or alternatively, to a maximal possible hash value). The last field (offset) in the entry indicates the absolute offset in bytes to the candidate into the fingerprinted object.

2. Initialize offsets min and max into the array M to 0. These variables point to the first and last elements of the array that are currently being used.

3. CutPoint(hash, offset) starts at step 800 in FIG. 8 and is invoked at each offset of the object to update M and return a result indicating whether a particular offset is a cutpoint.

The procedure starts by setting result = false at step 801.

At step 803, the procedure checks whether M[max].offset + h + 1 = offset. If this condition is true, execution continues at step 804 where the following assignments are performed: result is set to M[max].isMax, and max is set to max-1 % h. Execution then continues at step 805. If the condition at step 803 is false, execution continues at step 805.

At step 805, the procedure checks whether M[min].hash > hash. If the condition is true, execution continues at step 806, where min is set to (min-1) % h. Execution the continues at step 807 where M[min] is set to {isMax = false, hash =hash, offset=offset}, and to step 811, where the computed result is returned.

If the condition at step 805 is false, execution continues to step 808, where the procedure checks for whether M[min].hash = hash. If this condition is true, execution continues at step 807.

If the condition at step 808 is false, execution continues at step 809, where the procedure checks whether min = max. If this condition is true, execution continues at step 810, where M[min] is set to {isMax = true, hash =hash, offset=offset}. Execution then continues at step 811, where the computed result is returned.

If the condition at step 809 is false, execution continues at step 811, where min is set to (min+1) % h. Execution then continues back at step 805.

4. When CutPoint(hash, offset) returns true, it will be the case that the offset at position offset-h-1 is a new cut-point.

Analysis of Local Maximum Procedure

An object with n bytes is processed by calling CutPoint n times such that at most n entries are inserted for a given object. One entry is removed each time the loop starting at step 805 is repeated such that there are no more than n entries to delete. Thus, the processing loop may be entered once for every entry and the combined number of repetitions may be at most n. This implies that the average number of steps within the loop at each call to CutPoint is slightly less than 2, and the number of steps to compute cut points is independent of h.

Since the hash values from the elements form a descending chain between min and max, we will see that the average distance between min and max (|min – max| % *h*) is given by the natural logarithm of h. Offsets not included between two adjacent entries in M have hash values that are less than or equal to the two entries. The average length of such chains is given by the recurrence equation $f(n) = 1 + 1/n * \sum_{k<n} f(k)$. The average length of the longest descending chain on an interval of length n is 1 greater than the average length of the longest descending chain starting from the position of the largest element, where the largest element may be found at arbitrary positions with a probability of 1/n. The recurrence relation has as solution corresponding to the harmonic number $H_n = 1 + ½ + 1/3 + ¼ + ... + 1/n$, which can be validated by substituting H_n into the equation and performing induction on n. H_n is proportional to the natural logarithm of n. Thus, although array M is allocated with size *h*, only a small fraction of size ln(*h*) is ever used at any one time.

Computing min and max with modulus *h* permits arbitrary growth of the used intervals of M as long as the distance between the numbers remain within *h*.

The choice of initial values for M implies that cut-points may be generated within the first h offsets. The algorithm can be adapted to avoid cut-points at these first h offsets.

The expected size of the chunks generated by this procedure is around $2h+1$. We obtain this number from the probability that a given position is a cut-point.

Suppose the hash has m different possible values. Then the probability is determined by:

$$\sum_{0 \le k < m} 1/m\ (k/m)^{2h}.$$

Approximating using integration $\int_{0 \le x < m} 1/m\ (x/m)^{2h}\ dx = 1/(2h+1)$ indicates the probability when m is sufficiently large.

The probability can be computed more precisely by first simplifying the sum to:

$$(1/m)^{2h+1} \sum_{0 \le k < m} k^{2h},$$

which using Bernoulli numbers B_k expands to:

$$(1/m)^{2h+1}\ 1/(2h+1) \sum_{0 \le k < 2h} (2h+1)!/k!\ (2h+1-k)!\ B_k\ m^{2h+1-k}$$

The only odd Bernoulli number that is non-zero is B_1, which has a corresponding value of $-\frac{1}{2}$. The even Bernoulli numbers satisfy the equation:

$$H_{\infty}^{(2n)} = (-1)^{n-1}\ 2^{2n-1}\ \pi^{2n}\ B_{2n} / (2n)!$$

The left hand side represents the infinite sum $1 + (1/2)2n + (1/3)2n + \ldots$, which for even moderate values of n is very close to 1.

When m is much larger than *h*, all of the terms, except for the first can be ignored, as we saw by integration. They are given by a constant between 0 and 1 multiplied by a term proportional to h^{k-1} / m^k. The first term (where $B_0 = 1$) simplifies to $1/(2h+1)$. (the second term is $-1/(2m)$, the third is $h/(6m^2)$).

For a rough estimate of the expected slack consider streams s_1s_3 and s_2s_3. The last cut points inside s_1 and s_2 may appear at arbitrary places. Since the average chunk length is about $2h+1$ about ¼'th of the last cut-points will be within distance *h* in

both s_1 and s_2. They will contribute to cut-points at around 7/8 *h*. In another ½ of the cases, one cut-point will be within distance *h* the other beyond distance *h*. These contribute with cut-points around ¾*h*. The remaining ¼'th of the last cut-points in s_1 and s_2 will be in distance larger than *h*. The expected slack will therefore be around ¼ * 7/8 + ½ * ¾ + ¼ * ¼ = 0.66.

Thus, the expected slack for our independent chunking approach is 0.66 * C, which is an improvement over the prior art (0.85 * C).

There is an alternate way of identifying cut-points that require executing in average fewer instructions while using space at most proportional to *h*, or in average ln *h*. The procedure above inserts entries for every position 0..n-1 in a stream of length n. The basic idea in the alternate procedure is to only update when encountering elements of an ascending chain within intervals of length *h*. We observed that there will in average only be ln *h* such updates per interval. Furthermore, by comparing the local maxima in two consecutive intervals of length *h* one can determine whether each of the two local maxima may also be an *h* local maximum. There is one peculiarity with the alternate procedure; it requires computing the ascending chains by traversing the stream in blocks of size *h*, each block gets traversed in reverse direction.

In the alternate procedure (see FIGS. 10 and 11), we assume for simplicity that a stream of hashes is given as a sequence. The subroutine *CutPoint* gets called for each subsequence of length *h* (expanded to "horizon" in the Figures). It returns zero or one offsets which are determined to be cut-points. Only ln(*h*) of the calls to *Insert* will pass the first test.

Insertion into A is achieved by testing the hash value at the offset against the largest entry in A so far.

The loop that updates both `A[k]` and `B[k].isMax` can be optimized such that in average only one test is performed in the loop body. The case `B[1].hash <= A[k].hash` and `B[1].isMax` is handled in two loops, the first checks the hash value

against `B[l].hash` until it is not less, the second updates `A[k]`. The other case can be handled using a loop that only updates `A[k]` followed by an update to `B[l].isMax`.

Each call to *CutPoint* requires in average ln *h* memory writes to A, and with loop hoisting *h* + ln *h* comparisons related to finding maxima. The last update to `A[k].isMax` may be performed by binary search or by traversing B starting from index 0 in at average at most log ln *h* steps. Each call to *CutPoint* also requires re-computing the rolling hash at the *last* position in the window being updated. This takes as many steps as the size of the rolling hash window.

Observed Benefits of the Improved Chunking Algorithms

The minimal chunk size is built into both the local maxima and the filter methods described above. The conventional implementations require that the minimal chunk size is supplied separately with an extra parameter.

The local max (or mathematical) based methods produce measurable better slack estimate, which translates to further compression over the network. The filter method also produces better slack performance than the conventional methods.

Both of the new methods have a locality property of cut points. All cut points inside s3 that are beyond horizon will be cut points for both streams s1s3 and s2s3. (in other words, consider stream s1s3, if p is a position $\geq$ |s1|+horizon and p is a cut point in s1s3, then it is also a cut point in s2s3. The same property holds the other direction (symmetrically), if p is a cut point in s2s3, then it is also a cut point in s1s3). This is not the case for the conventional methods, where the requirement that cuts be beyond some minimal chunk size may interfere adversely.

Alternative Mathematical functions

Although the above-described chunking procedures describe a means for locating cut-points using a local maxima calculation, the present invention is not so limited. Any mathematical function can be arranged to examine potential cut-points. Each potential cut-point is evaluated by evaluating hash values that are located within the horizon window about a considered cut-point. The evaluation of the hash values is

accomplished by the mathematical function, which may include at least one of locating a maximum value within the horizon, locating a minimum values within the horizon, evaluating a difference between hash values, evaluating a difference of hash values and comparing the result against an arbitrary constant, as well as some other mathematical or statistical function.

The particular mathematical function described previously for local maxima is a binary predicate "_ > _". For the case where *p* is an offset in the object, *p* is chosen as a cut-point if $hash_p > hash_k$, for all k, where p-horizon $\leq k < p$, or $p < k \leq$ p+horizon. However, the binary predicate > can be replaced with any other mathematical function without deviating from the spirit of the invention.

The above specification, examples and data provide a complete description of the manufacture and use of the composition of the invention. Since many embodiments of the invention can be made without departing from the spirit and scope of the invention, the invention resides in the claims hereinafter appended.